ROB521: Assignment 1

Drini Kerciku

February 2023

1 Constructing a Graph through PRM

The graph was constructed by uniformly sampling the bounding region of the generated map and creating edges between each node and their neighbours within a radial distance of 1 meter, while checking for possible collisions. Due to using a uniform pseudo-number generator, clusters of samples would form in certain regions on the map, providing no additional information/coverage of the workspace.

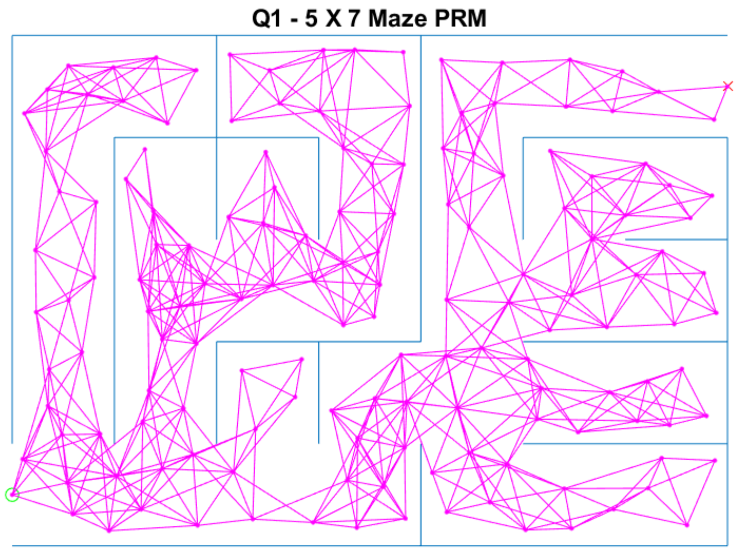

Figure 1: Sample of PRM constructed graph for a 4x7 grid - 0.21 seconds.

I filtered out points that were within a radial distance of 0.3 meters by parsing the nodes in the order they were sampled - the graph is created successfully from run to run with a high probability after determining a good combination of r_N and r_C which parameterize the ROI of neighbours and clusters respectively. An example of the constructed graph is depicted on Figure 1. The average time for constructing a graph is 0.35 seconds, which is reasonable for the given size of the labyrinth.

2 A* for the Shortest Path

I decided to implement the A* algorithm for traversing the graph and finding the shortest path, which has an average runtime of 0.21 seconds depending on the maze layout and number of nodes and connections after filtering the clustered samples and edge collisions with the walls - I am using a Manhattan distance metric as the admissible heuristic from node i to the goal and the Euclidean distance for the cost of edges. Given the variation of maps and the strategy-less sampling of possible robot locations, there are cases when the A* search takes longer than the construction of the graph and contradicts what was presented in lecture - it is a result of having a much more simplistic environment and obstacles that reduce the computation load when constructing the graphs. The solution to the graph on Figure 1 is displayed on Figure 2. In the graph presented under Section 1, the A* performance is significantly less than the algorithm designed to generate the graph.

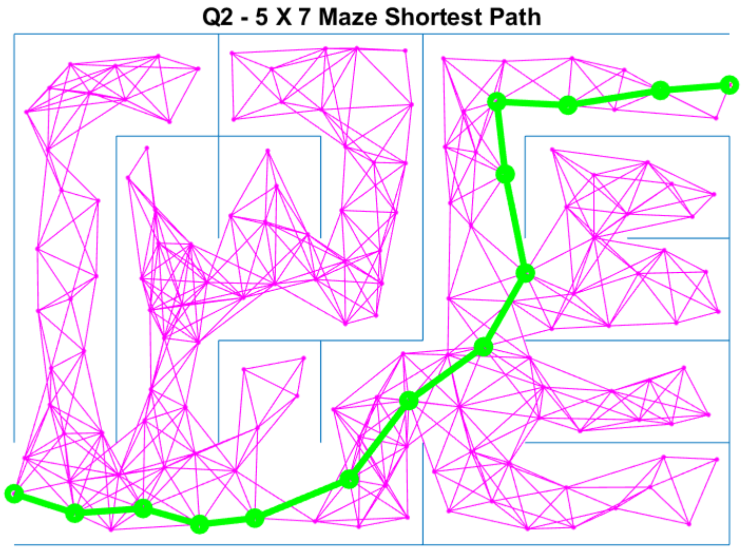

Figure 2: A* solution to the 4x7 labyrinth on Figure 1 - 0.0427 seconds.

3 Improving on Sampling

The method for sampling is not appropriate for covering large mazes of the sizes 40x40 or larger. I decided to take advantage of the environment structure and create a node located at the center of each cell, a tile of size 1x1 meters, which reduces the number of connections to a maximum of 4 per node. Given the size of the maze and the more 'strategic' approach to tackle graph creation, the A* algorithm takes the most amount of time in solving the planning problem. Other possible improvements for such scenario would include avoiding the compute of edge cost at every iteration and carrying it *a priori*.

In Section 4, the source code is configured to run only on 42x42 mazes as it manages to solve them consistently under the time limit with the pseudo-random number generator state being a function of the current time when the script is ran. It is important to note that the algorithm is able to solve up to 45x45 mazes within the limit of 20 seconds, but certain map variations prove to be more difficult due to the expansion of paths considered that may lead to the goal. I have included several successful runs in the following graphics.

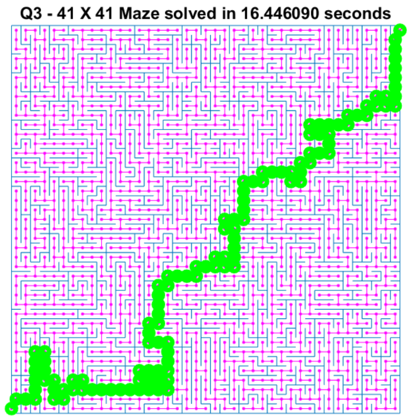

Figure 3: A* solution a 41x41 labyrinth.

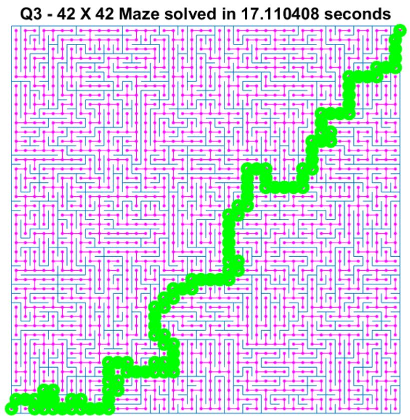

Figure 4: A* solution a 42x42 labyrinth.

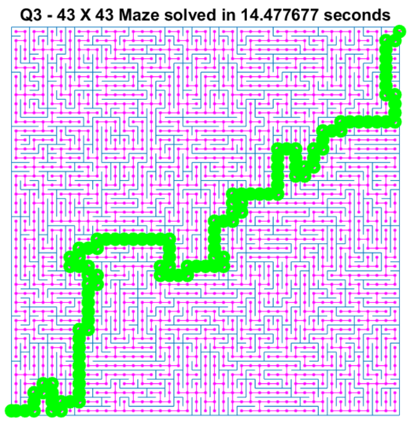

Figure 5: A* solution a 43x43 labyrinth.

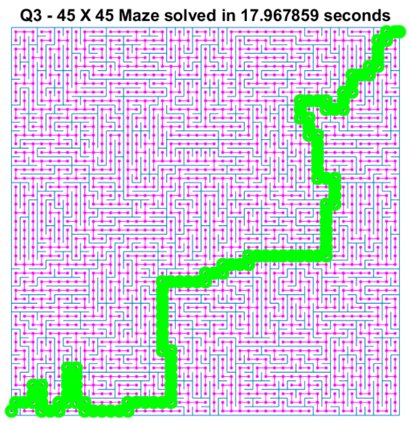

Figure 6: A* solution a 45x45 labyrinth.

4 Source Code

```
% ======
% ROB521_assignment1.m
% ======
%
% This assignment will introduce you to the idea of motion planning for
% holonomic robots that can move in any direction and change direction of
% motion instantaneously.  Although unrealistic, it can work quite well for
% complex large scale planning.  You will generate mazes to plan through
% and employ the PRM algorithm presented in lecture as well as any
% variations you can invent in the later sections.
%
% There are three questions to complete (5 marks each):
%
%    Question 1: implement the PRM algorithm to construct a graph
%    connecting start to finish nodes.
%    Question 2: find the shortest path over the graph by implementing the
%    Dijkstra's or A* algorithm.
%    Question 3: identify sampling, connection or collision checking
%    strategies that can reduce runtime for mazes.
%
% Fill in the required sections of this script with your code, run it to
% generate the requested plots, then paste the plots into a short report
% that includes a few comments about what you've observed.  Append your
% version of this script to the report.  Hand in the report as a PDF file.
%
% requires: basic Matlab,
%
% S L Waslander, January 2022
%
clear; close all; clc;

% set random seed for repeatability if desired
% rng(3,"v5uniform");
rng(sum(100*clock),"v5uniform")

% ==========================
% Maze Generation
% ==========================
%
% The maze function returns a map object with all of the edges in the maze.
% Each row of the map structure draws a single line of the maze.  The
% function returns the lines with coordinates [x1 y1 x2 y2].
% Bottom left corner of maze is [0.5 0.5],
% Top right corner is [col+0.5 row+0.5]
%

row = 5; % Maze rows
col = 7; % Maze columns
map = maze(row,col); % Creates the maze
start = [0.5, 1.0]; % Start at the bottom left
finish = [col+0.5, row]; % Finish at the top right

h = figure(1);clf; hold on;
plot(start(1), start(2),'go')
plot(finish(1), finish(2),'rx')
show_maze(map,row,col,h); % Draws the maze
drawnow;

```

```
% ====================================================
% Question 1: construct a PRM connecting start and finish
% ====================================================

% Using 500 samples, construct a PRM graph whose milestones stay at least
% 0.1 units away from all walls, using the MinDist2Edges function provided for
% collision detection.  Use a nearest neighbour connection strategy and the
% CheckCollision function provided for collision checking, and find an
% appropriate number of connections to ensure a connection from  start to
% finish with high probability.

% variables to store PRM components
nS = 500;  % number of samples to try for milestone creation
milestones = [start; finish];  % each row is a point [x y] in feasible space
edges = [];  % each row is should be an edge of the form [x1 y1 x2 y2]
edgeWallDis = 0.1;
neighbRad = 1;
sampleCount = 0;

disp("Time to create PRM graph")
tic;
% ------insert your PRM generation code here-------

% Uniformly Sample in 2D Region of the map and augment milestones
while sampleCount < nS

    % extract sample
    randX = rand(1,1);
    randY = rand(1,1);
    milX = row*randX + 0.5;
    milY = col*randY + 0.5;
    currPt = [milY, milX];

    % check if it is far enough from the edges of the maze and add to
    % milestones
    dist = MinDist2Edges(currPt, map);
    if dist > edgeWallDis
        milestones = [milestones; currPt];
    end

    sampleCount = sampleCount + 1;

end

% Preprocess the list of milestiones to remove big clusters within a small
% radius
removeId = [];
keepId = [];
clusterRad = 0.3;
milestones = unique(milestones, "rows", "stable");
coordX = milestones(:, 1);
coordY = milestones(:, 2);

for i = 1 : length(milestones)

    % check if this milestone has been already removed
    checkSum = sum(ismember(removeId,i));

    if checkSum == 0

```

```
        % find clustered milestones at i
        batchDist = sqrt((coordX - milestones(i,1)).^2 + (coordY - milestones(i,2))
    .^2);
        % find id-s that are within the cluster threshold
        ids = find((batchDist ~= 0) & (batchDist <= clusterRad));

        % remove clustered points
        if ~isempty(ids)
            for j = 1:length(ids)
                checkSum1 = sum(ismember(removeId,ids(j)));
                if checkSum1 == 0
                    removeId = [removeId;ids(j)];
                end
            end
        end

        % keep i-th element
        keepId = [keepId; i];

    end

end

% filter  entries
newMilestones = milestones(keepId,:);

% Create Graph using Nearest Neighbours Approach
coordX = newMilestones(:, 1);
coordY = newMilestones(:, 2);
nodeConnections = [0,0];

for i = 1 : length(newMilestones)

    % compute indices of nearest neighbours
    batchDist = sqrt((coordX - newMilestones(i,1)).^2 + (coordY - newMilestones(i,2))
    .^2);
    neighbId = find((batchDist ~= 0) & (batchDist <= neighbRad));

    % add collision free edges with the map
    if ~isempty(neighbId)

        for j = 1:length(neighbId)

            ptA = [newMilestones(i,1), newMilestones(i,2)];
            ptB = [newMilestones(neighbId(j),1), newMilestones(neighbId(j),2)];

            if (CheckCollision(ptA, ptB, map) == 0)

                % check if the edge already exists as we assume our graph
                % to be undirected
                pairCheck = [neighbId(j), i];
                [a, index] = ismember(nodeConnections, pairCheck, "rows");

                if sum(index) == 0
                    nodeConnections = [nodeConnections; i, neighbId(j)];
                    edges = [edges; ptA, ptB];
                end

            end

```

```
        end

    end

end

% ------end of your PRM generation code -------
toc;

figure(1);
plot(newMilestones(:,1),newMilestones(:,2),'m.');
if (~isempty(edges))
    line(edges(:,1:2:3)', edges(:,2:2:4)','Color','magenta') % line uses [x1 y1 x2 y2
    ]
end
str = sprintf('Q1 - %d X %d Maze PRM', row, col);
title(str);
drawnow;

print -dpng assignment1_q1.png


% ================================================================
% Question 2: Find the shortest path over the PRM graph
% ================================================================

% Using an optimal graph search method (Dijkstra's or A*) , find the
% shortest path across the graph generated.  Please code your own
% implementation instead of using any built in functions.

disp('Time to find shortest path');
tic;

% Variable to store shortest path
spath = []; % shortest path, stored as a milestone row index sequence


% ------insert your shortest path finding algorithm here-------

% Compute Cost to Go Using an Admissible Heuristic (h(x))
cost2Go = heuristicManhattan(newMilestones, finish);
% Initialize Search
gScore = inf(length(newMilestones), 1);
gScore(1) = 0;
fScore = inf(length(newMilestones), 1);
fScore(1) = cost2Go(1);
cameFrom = zeros(size(newMilestones, 1), 1);
openSet = [start];

nodeConnections(1,:) = [];

while ~isempty(openSet)

    % find the node with the least f in queueStart
    [q, qId] = findMinF(openSet, newMilestones, fScore);

    % compute path if we reached goal
    if sum(q - finish) == 0
        [a, idG] = ismember(newMilestones, finish, "rows");
        goalID = find(idG == 1);
```

```
        spath = reconstructPath(cameFrom, goalID);
        break
    end

    % remove q from the open set
    openSet(qId,:) = [];

    % find neighbours of q ( connections )
    nNeighb = getNeighbours(q, nodeConnections, newMilestones);
    nNSize = size(nNeighb);
    % get currGScore
    [a, index] = ismember(newMilestones, q, "rows");
    id = find(index == 1);
    currGScr = gScore(id);

    for i = 1 : nNSize(1)

        % compute tentative score and check
        iNeigh = nNeighb(i,:);
        currNScr = currGScr + edgeScore(q, iNeigh);
        % find current neighbour index
        [a, id_0] = ismember(newMilestones, iNeigh, "rows");
        idN = find(id_0 == 1);

        % update if possible
        if currNScr < gScore(idN)
           cameFrom(idN) = id;
           gScore(idN) = currNScr;
           fScore(idN) = currNScr + cost2Go(idN);
           % check if neighbour is in openSet
           [a, id_1] = ismember(openSet, iNeigh, "rows");
           checkSumN = sum(id_1);
           if checkSumN == 0
               openSet = [openSet; iNeigh];
           end

        end

    end

end

% ------end of shortest path finding algorithm-------
toc;

% plot the shortest path
figure(1);
for i=1:length(spath)-1
    plot(newMilestones(spath(i:i+1),1),newMilestones(spath(i:i+1),2), 'go-', '
    LineWidth',3);
end
str = sprintf('Q2 - %d X %d Maze Shortest Path', row, col);
title(str);
drawnow;

print -dpng assingment1_q2.png


% ================================================================
% Question 3: find a faster way
```

```
% ================================================================

% Modify your milestone generation, edge connection, collision detection
% and/or shortest path methods to reduce runtime.  What is the largest maze
% for which you can find a shortest path from start to goal in under 20
% seconds on your computer? (Anything larger than 40x40 will suffice for
% full marks)

row = 42;
col = 42;
map = maze(row,col);
start = [0.5, 1.0];
finish = [col+0.5, row];
milestones = [start; finish];  % each row is a point [x y] in feasible space
edges = [];  % each row is should be an edge of the form [x1 y1 x2 y2]

h = figure(2);clf; hold on;
plot(start(1), start(2),'go')
plot(finish(1), finish(2),'rx')
show_maze(map,row,col,h); % Draws the maze
drawnow;

fprintf("Attempting large %d X %d maze... \n", row, col);
tic;
% ------insert your optimized algorithm here------

% generate a single milestone per cell located on their centroid
cX = 1:1:col;
cY = 1:1:row;

for i = 1:length(cX)
    for j = 1:length(cY)
        milestones = [milestones; cX(i) cY(j)];
    end
end

% compute edges in a similar fashion to previous algorithm, now each cell
% is limited to 8 neighbours at most
coordX = milestones(:, 1);
coordY = milestones(:, 2);
nodeConnections = [0,0];
neighbRad = sqrt(2);

for i = 1 : length(milestones)

    % compute indices of nearest neighbours
    batchDist = sqrt((coordX - milestones(i,1)).^2 + (coordY - milestones(i,2)).^2);
    neighbId = find((batchDist ~= 0) & (batchDist <= neighbRad));

    % add collision free edges with the map
    if ~isempty(neighbId)

        for j = 1:length(neighbId)

            ptA = [milestones(i,1), milestones(i,2)];
            ptB = [milestones(neighbId(j),1), milestones(neighbId(j),2)];

            if (CheckCollision(ptA, ptB, map) == 0)

                % check if the edge already exists as we assume our graph
```

```
                % to be undirected
                pairCheck = [neighbId(j), i];
                [a, index] = ismember(nodeConnections, pairCheck, "rows");

                if sum(index) == 0
                    nodeConnections = [nodeConnections; i, neighbId(j)];
                    edges = [edges; ptA, ptB];
                end

            end

        end

     end

end

% Run A* Algorithm as is

% Compute Cost to Go Using an Admissible Heuristic (h(x))
cost2Go = heuristicManhattan(milestones, finish);
% Initialize Search
gScore = inf(length(milestones), 1);
gScore(1) = 0;
fScore = inf(length(milestones), 1);
fScore(1) = cost2Go(1);
cameFrom = zeros(size(milestones, 1), 1);
openSet = [start];

nodeConnections(1,:) = [];

while ~isempty(openSet)

    % find the node with the least f in queueStart
    [q, qId] = findMinF(openSet, milestones, fScore);

    % compute path if we reached goal
    if sum(q - finish) == 0
        [a, idG] = ismember(milestones, finish, "rows");
        goalID = find(idG == 1);
        spath = reconstructPath(cameFrom, goalID);
        break
    end

    % remove q from the open set
    openSet(qId,:) = [];

    % find neighbours of q ( connections )
    nNeighb = getNeighbours(q, nodeConnections, milestones);
    nNSize = size(nNeighb);
    % get currGScore
    [a, index] = ismember(milestones, q, "rows");
    id = find(index == 1);
    currGScr = gScore(id);

    for i = 1 : nNSize(1)

        % compute tentative score and check
        iNeigh = nNeighb(i,:);
        currNScr = currGScr + edgeScore(q, iNeigh);
```

```
        % find current neighbour index
        [a, id_0] = ismember(milestones, iNeigh, "rows");
        idN = find(id_0 == 1);

        % update if possible
        if currNScr < gScore(idN)
           cameFrom(idN) = id;
           gScore(idN) = currNScr;
           fScore(idN) = currNScr + cost2Go(idN);
           % check if neighbour is in openSet
           [a, id_1] = ismember(openSet, iNeigh, "rows");
           checkSumN = sum(id_1);
           if checkSumN == 0
               openSet = [openSet; iNeigh];
           end

        end

    end

end

% ------end of your optimized algorithm-------
dt = toc;

figure(2); hold on;
plot(milestones(:,1),milestones(:,2),'m.');
if (~isempty(edges))
    line(edges(:,1:2:3)', edges(:,2:2:4)','Color','magenta')
end
if (~isempty(spath))
    for i=1:length(spath)-1
        plot(milestones(spath(i:i+1),1),milestones(spath(i:i+1),2), 'go-', 'LineWidth
    ',3);
    end
end
str = sprintf('Q3 - %d X %d Maze solved in %f seconds', row, col, dt);
title(str);

print -dpng assignment1_q3.png

% ----------------------------------------------------------------------- %
% USER-DEFINED FUNCTIONS                                                  %
% ----------------------------------------------------------------------- %
function cost2Go = heuristicManhattan(milestones, goal)

    % initialize output
    cost2Go = zeros(length(milestones), 1);

    % evaluate Manhattan Distances
    for i = 1 : length(milestones)
        iPt = milestones(i,:);
            cost2Go(i) = sum(abs(iPt-goal));
    end

end
% ----------------------------------------------------------------------- %
function [q, qID] = findMinF(Queue, milestones, fScore)

    % find index of nodes in milestones
```

```
    sizeQ = size(Queue);
    stateId = [];
    for i = 1 : sizeQ(1)
        [a, index] = ismember(milestones, Queue(i,:), "rows");
        id = find(index == 1);
        stateId = [stateId; id];
    end

    % get respective fScores
    tempFSc = fScore(stateId);
    [score, idMin] = min(tempFSc);

    qID = idMin;
    q = Queue(idMin,:);

end
% ---------------------------------------------------------------------- %
function nNeighb = getNeighbours(curr, nodeConn, nodes)

     % initialize output
     nNeighb = [];

     % get node index in S
     [a, index] = ismember(nodes, curr, "rows");
     id = find(index == 1);

     % search for neighbours
     for i = 1:length(nodeConn)
         pair = nodeConn(i,:);
         if (pair(1) == id)
             nNeighb = [nNeighb; nodes(pair(2),:)];
         elseif (pair(2) == id)
             nNeighb = [nNeighb; nodes(pair(1),:)];
         end
     end

end
% ---------------------------------------------------------------------- %
function score = edgeScore(ptA, ptB)
    % compute Euclidean distance between to nodes
    score = sqrt(sum((ptA-ptB).^2));
end
% ---------------------------------------------------------------------- %
function path = reconstructPath(cameFrom, finishID)
    % backtrack the path to start
    path = finishID;
    prevID = 0;
    currID = finishID;
    while true
        prevID = currID;
        path = [path; cameFrom(prevID)];
        currID = cameFrom(prevID);
        if currID == 1
            break
        end
    end
end
```